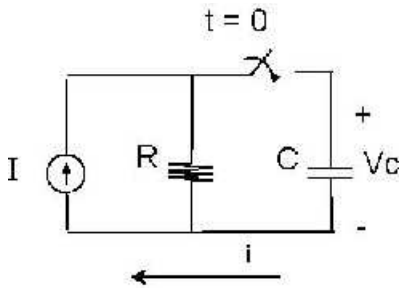
Basic Control System with matlab examples

Written by Chieh Wu

State Space Equations

It has been cutomary to characterize the internal state of a system using differential equations. Let's take this RC circuit for example.



We can describe this circuit using these equations.

$$\frac{dV_c}{dt} = \frac{-1}{RC} V_c + \frac{1}{C} I$$

$$V_c = \quad V_c$$

For analysis purpose, it is more useful to look at these equations in terms of the state space equation which has this form.

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

x characterizes the internal state of the system. In our circuit case, we are describing the voltage across the capacitor.

y is the output. (or the variable we are observing) In our circuit, the internal state happens to also be the output.

So here's the break down of the variables.

$$\dot{x} = \frac{\mathrm{dV_c}}{\mathrm{dt}} \qquad A = \frac{-1}{\mathrm{RC}} \qquad x = V_c \qquad B = \frac{1}{C} \qquad u = I$$

$$y = V_c \qquad C = 1 \qquad D = 0$$

The only difference between the differential equation and the state space equation is the way we are writing them. We are not changing anything in terms of the system.

Although we can write the differential equation in many forms, it has been discovered that writing the eqautions in the state space form has many benefits. First, it is difficult to study systems without a unified form. It is much easier for communication sake for everybody to just convert whatever form they have into a common form. (A,B,C,D)

Second, it is now possible to characterize ssystems of multiple variables. The example we just looked at is a single input and single ouput system. (SISO) It consists of only a single variable we are attemping to solve. But for a more complicated system that consists of many variables we are observing, A,B,C,D would convert into matrices instead of being just a variable.

For our previous example, we could be looking at two internal states instead of just one.

$$\frac{\mathrm{dV_c}}{\mathrm{dt}} = \frac{-1}{\mathrm{RC}} V_c + \frac{1}{C} I$$

$$\frac{\mathrm{di}}{\mathrm{dt}} = \frac{-1}{\mathrm{RC}} i$$

$$V_c = \qquad V_c$$

In this case, we can organize the equation into the state space equation using the matrix form.

$$\begin{pmatrix} \frac{\mathrm{dV_c}}{\mathrm{dt}} \\ \frac{\mathrm{di}}{\mathrm{dt}} \end{pmatrix} = \begin{pmatrix} \frac{-1}{\mathrm{RC}} & 0 \\ 0 & \frac{-1}{\mathrm{RC}} \end{pmatrix} \begin{pmatrix} V_c \\ i \end{pmatrix} + \begin{pmatrix} \frac{1}{C} \\ 0 \end{pmatrix} I$$

$$V_c \quad = \quad \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} V_c \\ i \end{pmatrix}$$

Once we have put the equations into the state space form, we can represent and study the system simply by:

$$\dot{x} = A x + B u$$

$$y = C x + D u$$

Solutions to the state space equation

The solution to the state space equations in the continuous time domain given that x starts at x_0 are

$$x = e^{At}x_0 + \int_0^t e^{A(t-\tau)}\mathrm{B}u(\tau)\,d\tau$$

where the zero input response is

$$x = e^{At}x_0$$

the zero state response is

$$x = \int_0^t e^{A(t-\tau)}\mathrm{B}u(\tau)\,d\tau$$

We are able to combine the zero state and zero input responses due to the superposition principle. For all $t \geq 0$, The corresponding output trajectory y is given as:

$$y(t) = \mathrm{C}x(t) + D\,u(t)$$

$$y(t) = C[e^{At}x_0 + \int_0^t e^{A(t-\tau)}\mathrm{B}u(\tau)\,d\tau] + D\,u(t)$$

Linearization

In the above example, we were working with a linear system. What if the system we are working with is some complicated non-linear system.

$$\dot{x} = f(x)$$

This type of non-linear system would be very difficult to analyze, so it is generally converted into a linear form using linearization.

The concept of linearization is difficult to grasp unless an example is provided. Let's say we have a one dimensional function.

$$f(x) = 3(x-1)^2 - 1$$

According to the Taylor's Theorem, any function can be broken down into this series.

$$f(x) = f(x_{\text{eq}}) + f'(x_{\text{eq}}) * (x - x_{\text{eq}}) + \frac{f''(x_{\text{eq}})}{2!}(x - x_{\text{eq}})^2 + ... + \frac{f^{(n)}(x_{\text{eq}})}{n!}(x - x_{\text{eq}})^n$$

So if we use our example, it woud become

$$\begin{aligned} f(x) &= 3(x-1)^2 - 1 \\ f'(x) &= 6x - 6 \\ f''(x) &= 6 \\ f'''(x) &= 0 \end{aligned}$$

$$\begin{aligned} f(x_{\text{eq}}) &= 3(x_{\text{eq}} - 1)^2 - 1 \\ f'(x_{\text{eq}}) * (x - x_{\text{eq}}) &= (6x_{\text{eq}} - 6)(x - x_{\text{eq}}) \\ \frac{\text{f"}(\text{x}_{\text{eq}})}{2!}(\text{x - x}_{\text{eq}})^2 &= \frac{6}{2!}(x - x_{\text{eq}})^2 \end{aligned}$$

So $f(x)$ according to taylor series could be rewritten as

$$f(x) = \{3(x_{\text{eq}} - 1)^2 - 1\} + \{(6x_{\text{eq}} - 6)(x - x_{\text{eq}})\} + \{\frac{6}{2!}(x - x_{\text{eq}})^2\} + 0 + 0 + ... + 0$$

If you don't believe that this function above is exactly the same as

$$f(x) = 3(x-1)^2 - 1$$

You should try to manipulate the algebra to see if they are the same. You can use any x_{eq} and the two functions will still be exactly the same. If you go solve the algebra, you will realize that the value of x_{eq} really doesn't matter because it wil cancel out during the manipulation.

Also notice that there are 3 sections to this equation, the term with the first derivatvie, second derivative, and third derivative. Together they add up to the the function $f(x)$. If we take 2 out of 3 parts and use that, we would end up with another function that is relatively close to $f(x)$ but not exactly alike.

During linearization, we first pick the perfect x_{eq} so that the first term would equal to zero. Our goal is to get the taylor expansion into this form.

$$f(x) = \{(6x_{eq} - 6)(x - x_{eq})\} + \{\tfrac{6}{2!}(x - x_{eq})^2\}$$

Notice that the first term is gone because we have picked an equalibrium point such that

$$0 = 3(x_{eq} - 1)^2 - 1$$

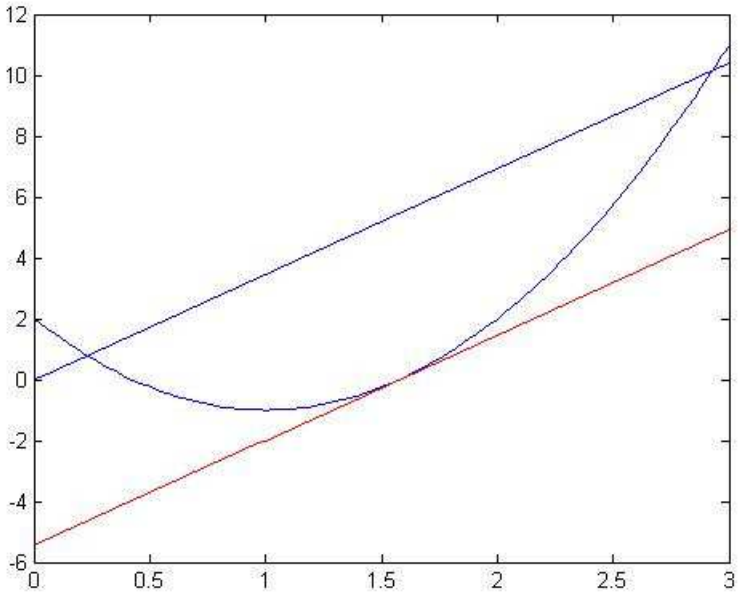
The next thing we do is to take the first term and forget about all other terms so that we end up with

$$f(x) = \{(6x_{eq} - 6)(x - x_{eq})\}$$

For the sake of this example, I have calculated ahead of time what the equalibrium point is.

$$x_{eq} = 1.5774$$

Let's take a look what we are doing by keeping only 1 term out of 3.



The parabola is the original function while the red line is the linearized version of the function. Notice that around the equalibrium point 1.5774, the error is zero. The error grows larger and larger when we get further apart from the equalibrium point. The straight blue line parallel to the red line is the linearization before shifting the line to the equalibrium point. As you can see, if you try to guess the values near the equalibrium point using the blue line, you will be off.

Luckily, in most engine applications, we don't need to have the result to be exact. So a little error is normally tolerable. For our case, if we only care about x from 1 to 2, our equalibrium point will give us a relative small error.

The exercise we have just gone through is an example of a single variable linearization. If we are dealing with a multivariable system, the general procedure is still the same. We are still finding the equalibrium point where the first term becomes zero and we are still using only the second term.

The only difference is that instead of dealing with one numbers, we are now dealing with matrices. Let's look at an example.

Define $f: \mathbb{R}^2 \to \mathbb{R}^2$

$$f(x) = \begin{pmatrix} x_1 + x_2^2 \\ x_2 + 1 \end{pmatrix}$$

If we want to put this function into the state space equation, we would need to linearize it. To do so, we would find the Jacobian of the matrix which is defined as:

$$\frac{\partial f}{\partial x} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & & & \\ \frac{\partial f_n}{\partial x_1} & \cdots & \cdots & \frac{\partial f_n}{\partial x_n} \end{pmatrix}$$

The concept of Jacobian is analogous to derivatives for matrices. Instead of just taking the derivative of a veriable, we are find the partial derivative of each function. So for our example, we would end up with.

We first find the equalibrium point where

$$0 = x_1 + x_2^2$$

$$0 = x_2 + 1$$

The equalibrium point here is

$$x_1 = 0$$

$$x_2 = 0$$

And the Jacobian is

$$A = \begin{pmatrix} 2 & 2x_2 \\ 0 & 1 \end{pmatrix} |_{x_{1,2}=0} = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}$$

The equation for a line looks like this

$$y = a(x - x_{\text{eq}})$$

or in our case

$$y = A(x - x_{\text{eq}})$$

You can see the jacobian as the slop of the line and the equalibrium point as how far to shift the linearized line. From our example with the parabola, the blue line would be Ax while the red line is $A(x - x_{\text{eq}})$. In linearization, we look only at Ax for us to fit into the state space equation model. So when we solve for x, we are really solving for $(x - x_{\text{eq}})$. It is very important therefore that after we have calculate the x value, it should be shifted (added to) by x_{eq}.

Very often $x_{\text{eq}} = 0$ as we have in this case. When that is the case, we don't need to shift. But when the equalibrium point is not zero, our calculation will be off by x_{eq}.

In summary, when we encounter non-linear system. It is necessary to linearize the system if we want to study the system using the state space equations. Towards this mean, here are the steps for us to linearize the system.

1. We first expand out the functions using the taylors series

   (this requires that you use jacobians to find the derivatives for the matrices.)

2. We find the equalibrium piont x_{eq} such that the first term is cancelled to zero.

3. We cancel out every term except the first derivative term, and use the first term as our estimate of the actual system.

4. You can now use the Jacobian to form the state space equation.

Linearization Using Matlab

Let's do a quick example together.

Let's say we have this non-linear system.

$$\begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \end{pmatrix} = \begin{pmatrix} -\mu & \omega \\ -\omega & -\mu \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} - (x_1^2 + x_2^2) \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \end{pmatrix} w + \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} u$$

$$y = \begin{pmatrix} 1 & 0 \end{pmatrix} x + v + y_b$$

Where

w and v are white noise

y_b is an unknown constant bias

$u = 0$

$x_0 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$

The first thing we need to do is to write a data .m file which initializes all the parameters.

```
clc;
u = -1/10;
w = 10;
x0 = [0,1];
sysdata = struct('nx',2,'nu',2,'ny',1);
sysdata = setfield(sysdata, 'x0', x0);
sysdata = setfield(sysdata, 'u', u);
sysdata = setfield(sysdata, 'w', w);
```

.....

In this previous section, we are basically setting up a structure in matlab called sysdata. The fields we set:

nx = number of states 2, because we have x1 and x2

nu = number of inputs, we have u1 and u2

ny = the number of output which we have just y

I normally also write the code in this same file that simulates the functions

.....

```
[A,B,C,D] = linmod('funct')
[t,x,y] = sim('funct',10);
plot(t,y);
hold [t,x2,y2] = sim('linearized',10);
plot(t,y2,'red');
```

.....

In the code above, we linearized the non-linear system and created the A,B,C,D matrices. Then we simulate the non-linear system and plot the output of the non-linear system. Once we have plotted out the non-linear system, we also simulate the linearized system for 10 seconds. Then we plot out the linearized system.

linmod = linearizes the non-linear system we have

funct = name of the simulink file we will write

sim = actually simulates the function for 10 seconds

The next matlab file you need to write is where you actually define the function. I happens to call my function.m but you can call it anything you want.

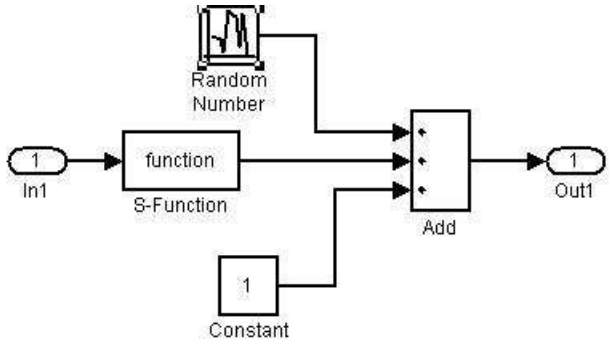
.....

```
function [sys,x0,str,ts] = Feed(t,x,u,flag,sysdata)
switch flag,
case 0,                % set initial state,state space dimensions,etc
  sizes = simsizes;
  sizes.NumContStates = sysdata.nx;          % number of continuous states
  sizes.NumDiscStates = 0; % number of discrete states
  sizes.NumOutputs = sysdata.ny;             % number of outputs
  sizes.NumInputs = sysdata.nu;              % number of inputs
  sizes.DirFeedthrough = 0;
  sizes.NumSampleTimes = 1;
  sys = simsizes(sizes);
  x0 = sysdata.x0; % initial state
  str = [ ];
  ts = [0 0];
case 1,                              %  calculate state derivatives
  sys = f(x,u,sysdata);                    % function where xdot is calculated
case 3,                               % calculate outputs
  sys = [1 0]*x;
case {2,4,9}, % remaining cases
sys = [ ];
otherwise
  error(['Unhandled flag = ',num2str(flag)]);
end

function [xdot] = f(x,u,sysdata)
  mu = sysdata.u;
```

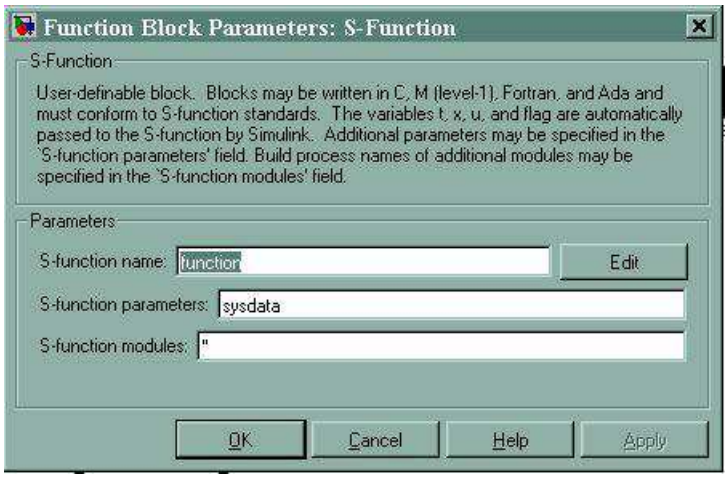
```
w = sysdata.w;
xdot = [-mu*x(1) + w*x(2)-x(1).3-x(1)*x(2).2+u(1)+u(2);
          -w*x(1) - mu*x(2)-x(2)*x(1).2-x(2).3+u(1)+u(2)];
```

.....

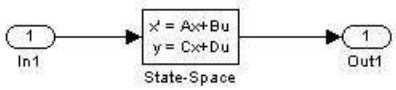
Here's a picture of what the simulink should look like.



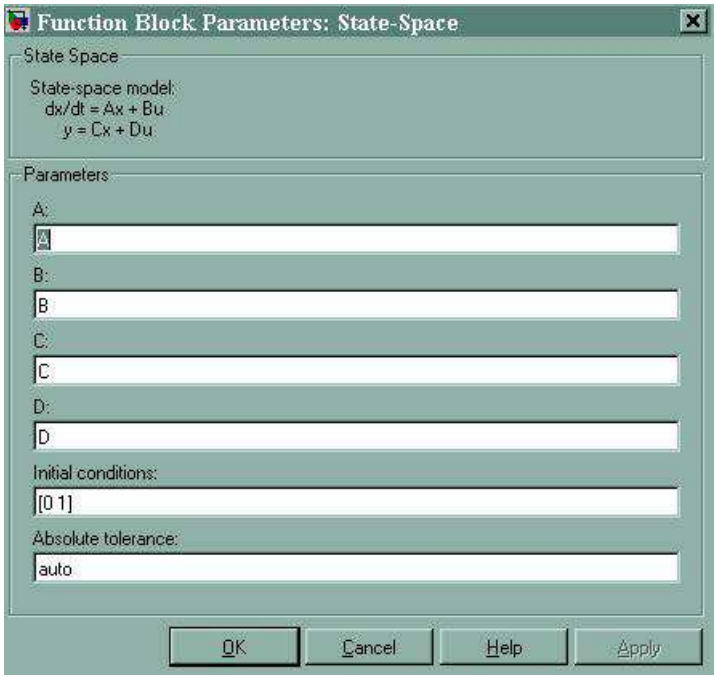
When you double click on the S-function block, make sure that the s-function name is the matlab .m file where you defined the function. In my case, I have defined it to be function. sysdata is the structure that includes the all the parameters you defined in the original data file.



Lastly, you also want to simulate the linearized version, so you would need to build another simulink mdl file that looks like this. I called this file linearized.mdl.



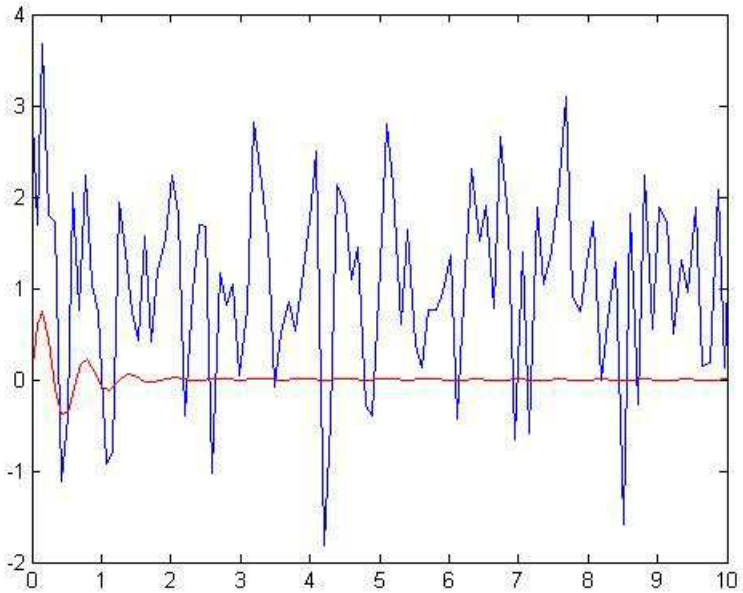
Remember from the first data mfile, when I linearized the non-linear system, I used the function linmod which returned A,B,C,D. They will be the input into the matrices when you double click on the state-space block.



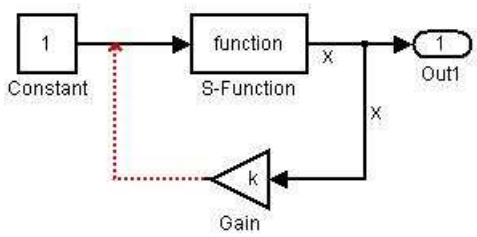
Here's the command from the data file that simulated this .mdl file

```
[A,B,C,D] = linmod('funct');
[t,x2,y2] = sim('linearized',10);
```

Once you have finally, build all the files, here is the simulate of both linear and non-linear system. The Blue signal is the non-linear simulation while the red is the linear.



The purpose of the Linear Quadratic Regulator is to provide a feedback to the system so that the system won't blow up.



This in theory sounds great, but there are many problems we encounter with LQR. First, notice that the output is x instead of y. This means that the output y must equal to x. If you remember from the previous chapter on LQR, one of the assumption we made was that the input u was equal to kx.

$$u = kx$$

This immediately gave us a contrain in terms of the systems that we can work with. We can only work with these kind of system.

$$\dot{x} = Ax + Bu$$

$$y = x$$

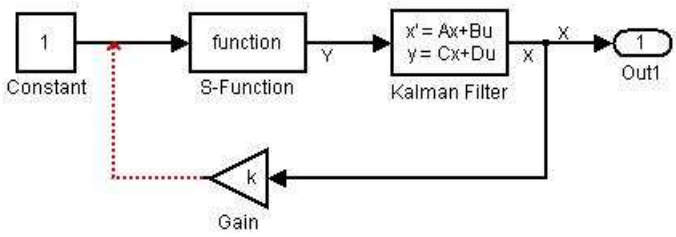
Unfortunately, the majority of the system that we will run into have the form.

$$\dot{x} = Ax + Bw$$

$$y = Cx + Dw$$

Where w is the Gaussian white noise.

This is where Kalman Filter comes in. It basically takes the output y from the system and estimate the best internal state x. Here's a graphical way to look at it.



The general idea starts with the concept of error. We are trying to find the best x', such that it is almost equal to the internal state x. So the difference between what we are guessing and the real thing is what we define as the error, ε.

$$\varepsilon = x - x'$$

We are working with this type of system

$$\dot{x} = Ax + Bw$$
$$y = Cx + Dw$$

We will estimate the internal state using this equation

eq1

$$\dot{x'} = Ax' + L(y - Cx')$$
$$y' = x'$$

also written as

eq2

$$\dot{x'} = (A - LC)x' + Ly$$
$$y' = x'$$

There are several things you should take note of with the above 2 equations.

First:

Notice how the input u position is now y. If you look at the diagram above, you would notice that the input into the Kalman filter is the output of the system we are dealing with, y.

Second:

Notice that the output of the Kalman Filter is the same thing as the estimated x'. This means that our estimation of x is going to come out of the Kalman filter.

Third:

Notice that we already know (A,C), they are from the original system.

Forth

Notice that we don't know L. This is the only term in the Kalman filter we need to find. Our goal is to find the optimum L such that we minimize the error between x and x'.

To minimize the error, our goal is then to minimize the variance (covariance) of error ε. Here's what we are trying to solve.

$$\min_{L} \quad \mathrm{Cov}[\varepsilon]$$

We start with this equation

$$\varepsilon = x - x'$$

$$\dot{\varepsilon} = \dot{x} - \dot{x'}$$

From the system and equation 1, we can replace $\dot{x}$ and $\dot{x}'$

$$\dot{\varepsilon} = Ax + Bw - [Ax' + L(y - Cx')]$$

$$\dot{\varepsilon} = A(x - x') + Bw - Ly + LCx'$$

Now we can also replace y from the state equation

$$\dot{\varepsilon} = A(x - x') + Bw - L(Cx + Dw) + LCx'$$

$$\dot{\varepsilon} = A(x - x') - LC(x - x') + (B - LD)w$$

We can replace $(x - x')$ with ε

$$\dot{\varepsilon} = A\varepsilon - LC\varepsilon + (B - LD)w$$

$$\dot{\varepsilon} = (A - LC)\varepsilon + (B - LD)w$$

From the first section, we know that after we solve for ε we have

$$\varepsilon = e^{(A-LC)t}\varepsilon_0 + \int_0^t e^{(A-LC)(t-\tau)}(B - LC)w(\tau)\,d\tau$$

Next we solve for the covariance of the error at $t = \infty$

$$\mathrm{Cov}[\varepsilon] = E[\varepsilon\varepsilon^*]$$

The Gaussian noise is the only non-deterministic term for the expectation. Where W is the covariance of the noise.

$$\int_0^t \int_0^t e^{(A-LC)(t-\tau)}(B-LC)E[w(\tau)w(\tau)^*](B-LC)^* e^{(A-LC)^*(t-\tau)}d\tau d\tau$$

$$\int_0^t \int_0^t e^{(A-LC)(t-\tau)}(B-LC)W(B-LC)^* e^{(A-LC)^*(t-\tau)}d\tau d\tau$$

The solution to this equation is equal to P, the solution for the lyapunov equation

$$(A-LC)P+P(A-LC)^*+(B-LC)W(B-LC)^*=0$$

For the sake of this problem, let's assume that W = 1

After rearranging the algebra, we have

$$AP+PA^*+BB^*-L(CP+DB^*)-(CP+DB^*)^*L^*+LDD^*L^*=0$$

Assuming that DD^* is invertable, we can get the previous form to look like this.

$$AP+PA^*+BB^*-[L-(CP+DB^*)^*(DD^*)^{-1}](DD^*)[L-(CP+DB^*)^*(DD^*)^{-1}]^*-(CP+DB^*)^*(DD)^{-1}(CP+DB^*)=0$$

From this point, we note this section of the formula

$$[L-(CP+DB^*)^*(DD^*)^{-1}](DD^*)[L-(CP+DB^*)^*(DD^*)^{-1}]^*$$

Notice that this value will always be positive, so if our goal is to minimize the value of P, we want this term to be as small as possible. The smallest term possible is zero so we want to find the L such that

$$L-(CP+DB^*)^*(DD^*)^{-1}=0$$

or

$$L=(CP+DB^*)^*(DD^*)^{-1}$$

Once that term is put to zero, the rest of the formula becomes.

$$AP+PA^*+BB^*-(CP+DB^*)^*(DD)^{-1}(CP+DB^*)=0$$

This is called the Kalman filter Riccati equation. From this point, we can solve for P.

Using Kalman Filter to observe the internal state

We have mentioned before that we can use Kalman Filter to "guess" the internal state of a system. Just because you can see the output of a system, it doesn't mean that you know what is happening inside the system. The kalman filter allows you to take the output of the system and tell you if what's inside is blowing up or calm down.

In this section, we are going to take an example system and add a bias. If you look at the function below, notice that the output y is the combination of x and a constant value y_b. So the kalman filter will look at y and tell you what y_b as well as x_1 and x_2 are.

I want to also mention that since we have made up this system, we could do what ever we want with y_b. As a matter of fact, I am going to set it as 5. But keep in mind that the Kalman filter we are about to design will have no clue what y_b is, but will still guess it.

So here's the system.

$$\begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \end{pmatrix} = \begin{pmatrix} -\mu & \omega \\ -\omega & -\mu \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} - (x_1^2 + x_2^2) \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \end{pmatrix} w + \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} u$$

$$y = \begin{pmatrix} 1 & 0 \end{pmatrix} x + v + y_b$$

Where

w and v are white noise

y_b is an unknown constant bias

$u = 0$

$x_0 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$

We are going to observe the zero input response.

The first thing we want to do is to linearize it.

$$\begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \end{pmatrix} = \begin{pmatrix} -\mu & \omega \\ -\omega & -\mu \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \end{pmatrix} w + \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} u$$

Since y_b is also something we want to track, we need to make it into a state.

$$\begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{y}_b \end{pmatrix} = \begin{pmatrix} -\mu & \omega & 0 \\ -\omega & -\mu & 0 \\ 0 & 0 & e^{-4} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ y_b \end{pmatrix} + \begin{pmatrix} 1 & 1 & 1 \\ 1 & -1 & 1 \\ 0 & 0 & e^{-4} \end{pmatrix} \begin{pmatrix} u \\ w \end{pmatrix}$$

$$y = \begin{pmatrix} 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ y_b \end{pmatrix} + \begin{pmatrix} e^{-4} & e^{-4} & e^{-4} \end{pmatrix} \begin{pmatrix} u \\ w \end{pmatrix} + v$$

You might have noticed that we didn't make $\dot{y}_b = 0$. This is for the kalman filter sake, we must have matrices with full rank. So instead of having zero, we just use a very small number. By changing this number, it actually determines how quickly we can track the bias.

The next thing we need to do is to find the kalman gain L, we can go through the math by hand, but luckily matlab does it for us with these commands.

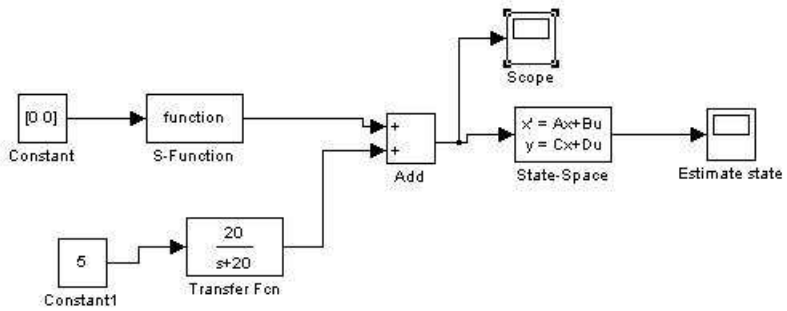
```
A = [-u w 0; -w -u 0; 0 0 0];
B = [1 1.0000 1.0000 1 -1.0000 1.0000 0 0 2*exp(-1)];
C = [1.0000 0 1];
D =[exp(-1) exp(-1) 2*exp(-1)];

sys = ss(A,B,C,D);
[kesk, L, P] = kalman(sys,1,1);
```

The first command ss, defines a system for us, while kalman help us find the kalman gain. Out of the three outputs kesk, L, and P, we only need the kalman gain L.

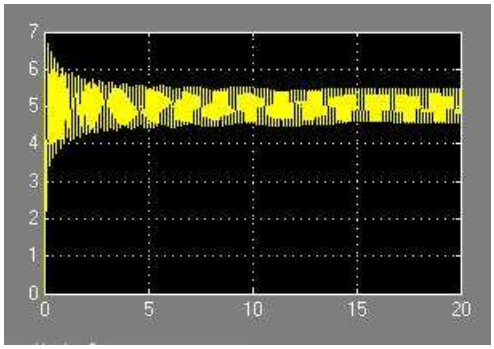
Once we have the kalman gain, we can find the output using a state space block with these state space equations.

$$\dot{x'} = (A - LC)\,x' + Ly$$
$$x' = x'$$

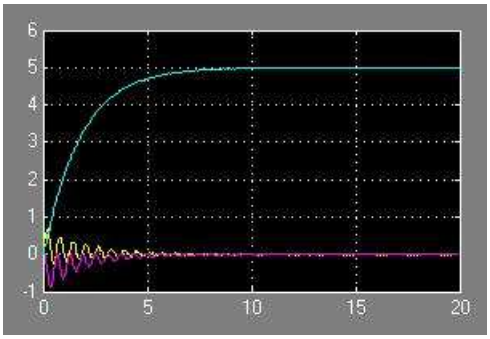
Here is a picture of what the simulink block should look like.



Take a look at the blocks. We have already covered how S-functions work from previous examples I have written. Notice how we have added a bias of 5 at the output. So here's what the Kalman filter sees. (Taken from scope)



Now, here's what the kalman filter thinks the state as well as our bias should be.



Take a while guess which line is tracking the bias 5, and which are tracking the other states. Now, remember how we introduced the fake noise before for y_b instead of using zero. By controlling that number, it allows us to track it faster or slower. The small the number the slower it tracks. If i make that number something like e^{-3}, it would take much longer to congerge to 5 than the picture above.